Technical Manual

Arana Charlebois, Tristan
Carvalho, Cam
Corbett, Cole
Kassie, Turner
Pauls, Andrew
Wallace, Jordan

Cosc 4P02
Dr. Naser Ezzati-Jivan

April 27, 2025

Contents

Preface

For the complete final report for our Cosc 4P02 project, please refer to the file named *Cosc-4p02-Final-Report.pdf* within the following repository https://github.com/BrockU-4P02-Logistics-System/Frontend/tree/main/Documentation. The following document is designed to represent a section of the overall final report.

1 Introduction

The following document has been produced to explain the technical intracarcies of our logistics system. It outlines the architecture of the overall system, in the form of diagrams and explanations of key components of each subsystem. This document has been divided into four main sections, namely, system overview, the backend, middleware, frontend. All repositories for specific subsystems will be linked within that respective section.

2 System Overview

We begin by outlining the system overview, from a high level. The roles of each subsystem within the complete system can be viewed as follows;

- **Backend:** Used for vehicle routing and computations regarding the ordering of nodes to travel to. This is done for both single drivers and multiple drivers.
- **Frontend:** Displayed routing information to the user. Connects to the database for information regarding user sessions, accounts, and information tied to accounts.
- **Middleware:** Used for the passing of information between the front and backend of the system, particularly, taking a general list of locations from the frontend of the system and converting them to (x, y) pairs for the backend and returning these pairs as locations to the frontend. Also, stores routing requests in a queue to be consumed by the backend.

Consider the component diagram, which models the behavior of users interacting with the system. Also note the connections between the DB, dashboard and backend (see figure 1).

Interactions can be seen as follows:

1. Users begin by accessing the landing page which they use to proceed to authorization. The authorization process is verified by comparing the form data with account data stored in the database.
2. Once authorized user are directed to the dashboard. From the dashboard, they can access the other related pages from here, namely, saved routes, fleet details, and billing pages. The dashboard is the main page that users can route vehicles on.

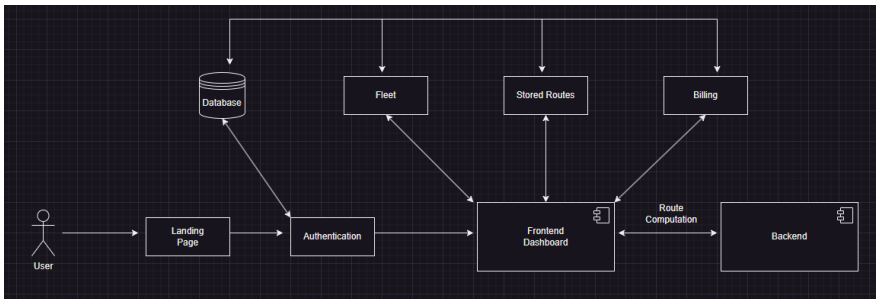

Figure 1: User Interaction with System

3. All pages concerning the storage of user data have functions that interact with the database.
4. Routing of vehicles computed by the backend has its message passing facilitated by RabbitMQ channels. Users input route information, and receive the computed result from the backend, which is shown in the bidirectional relationship in the above diagram 1.

3 Backend

https://github.com/BrockU-4P02-Logistics-System/Backend. The backend of the system is concerned with the actual computation of routes, for both cases of single and multiple drivers. The single driver case makes use of our own coded GA, whereas multiple drivers makes use of the Google-OR tools *vehicle routing* library. We begin with the following component diagram of the backend, illustrating the main elements of the subsystem (see figure 2).

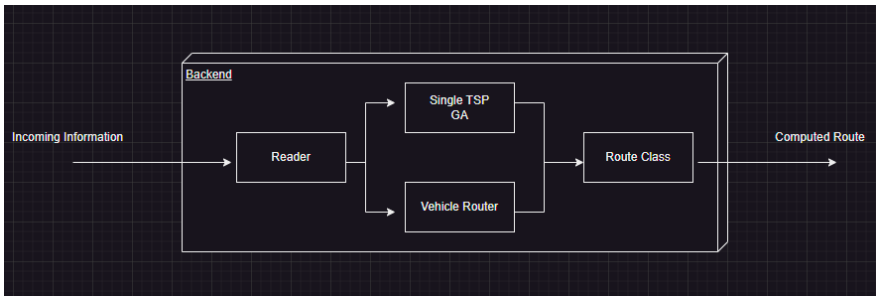

Figure 2: Backend Component Diagram

At a high level, we outline functionality as follows:

1. The backend receives a formatted file from the front end with information on the route options along with a list of coordinates. Note that the data

structure has been altered by RabbitMQ.
2. Reader parses the data, and determines the other classes to call.
3. Information is passed to the instance of the GA in the case of single driver routing or computes a distance matrix along with creation of an instance of Google-OR Tools vehicle router.
4. Either instances compute the resulting route, by reordering the nodes being traveled too.
5. Finally, based on the ordering, a final route is the written to the output file to be read by the frontend (Once again passed by RabbitMQ). This file contains additional information from the one that was read particularly, assigning a driver to each node.

3.1 Genetic Algorithm

The GA is concerned with routing one driver, deploying the use of standard GA methods and concpets, such as mutation, crossover, selection, elitism, etc. This code was first developed in the *GeneticAlgorithm.java* class and was then refactored to be the *GeneticAlgorithm2.java* class. The input is a general list of coordinates, in the form of a geojson, and the output is the best ordering determined by the GA, also in a geojson. While this GA follows a standard GA format of iterating (seen in *mainLoop()*), performing crossovers, and mutation of individuals, we now list the problem specific aspects of each of the GA's methods to aid in outlining functionality.

- Individuals store an ordered list of locations, each location representing a stop.
- Fitness of individuals is the driving time between each stop, calculated by Graph Hopper, all summed up for complete driving time.
- Crossover is an ordered crossover, while not touching the first location as this is fixed.
- Mutation is simply swapping two locations in an individual, not allowing for the first location to be swapped.

3.2 Multi TSP

The manner in which multi TSP is solved is by using the Google-Or Tools library, particularly, solved with the *VehicleRouter.java* class where Or-Tools is used. The input for the Google-Or Tools is a distance matrix, that is filled with the travel times between all nodes, and returns a viable routing. We enumerate the internal steps for solving multi TSP.

1. Read in the locations to use for routing.
2. Compute distances between nodes, and add to adjacency matrix.
3. Pass the matrix to Or-Tools for assignment of drivers to nodes, and a drivers order within its set of nodes.
4. With computed route, call route class for sending the information back to the frontend of the system, in standardized format.

The actual object performing the routing is the *routingModel*, with the input being the *routingIndexManager* object. The *routingIndexManager* is concerned basic configuration of the routing problem such as the number of vehicles and returning to the starting location. In the case of our system, the elements of the *routingIndexManager* is populated with the boolean flags being sent from the frontend of the system.

3.3 Dependencies

We list the two main libraries used within the backend of the system and explain their significance.

- **Graph Hopper:** The graph hopper library is a key dependency of the backend of the system, as it is used for the travel time computation. In general, Graph hopper can be used for additional information, such as distance and finding the actual path between stops. Using the North America open street map file, graph hopper can determine routes of travel for different types of modes of transportation (we make use of automobiles).
- **Google Or-tools:** Or-tools is being deployed for solving the multi driver routing problem. The input for the solution as mentioned is a weighted adjacency matrix, with driving times (computed by Graph hopper) being the weights of edges. OR-Tools is computing the assignment of each location to a driver, then finding the order of stops for each of the drivers.

3.4 Hosting of the Backend

The backend of the system is hosted on a 64-GB ram Oracle server in the form of consumers (see section 4). We have the capacity to work with 4 consumers, each of which operate as a backend instance. These instances of the backend are hosted beside the RabbitMQ queue. Since these components are hosted side-by-side, latency is minimized from a user perceptive due to less time spent message passing. By hosting the RabbitMQ queue and the backend locally, we save a network hop to and from the backend. Connections to the frontend are made through RabbitMQ too.

4 Middleware

The middleware for handling instances of the backend lives side by side the backend code on the same Oracle cloud server. After a user makes a request to compute a route, a connection is established between the frontend web server and the request queue. This functionality is specific to the *process* file found by following https://github.com/BrockU-4P02-Logistics-System/Frontend/tree/main/app/api/process. The request queue is a data structure that stores frontend requests, and these requests are handled individually by a consumer with the backend (either the GA or Or-Tools). The queue stores route requests from different machines and abiding by the FIFO principle passes these

requests when there is an available consumer. It is easy to scale up the total number of consumers. It is possible for the consumers to be dispersed across a cluster or network of servers.

When there is a request at the front of the queue, our code assigns it to a consumer, that consumer then runs instance of the backend, which remains warm, ready to service requests. To perform this, an object called *logisticApplication* is instantiated, and this object contains the backend. *LogisticsApplication* passes the parameters with the route data from the request that it got from the front of the queue.

Once the backend has completed routing, the results need to be returned to the frontend of the system. Message passing between the frontend and the RabbitMQ queue is done through a RabbitMQ channel. The RabbitMQ channel is a secure message passing protocol that passes strings across the network. The strings being passed in our application are location objects and route parameters. Our frontend deals with location objects and the backend takes coordinate pairs. Between the front and backend, the middleware must facilitate the conversion of coordinates to locations and vice versa. A Google Maps API call performs this conversion, knows as geocoding. A set of locations or (x, y) pairs are stringified before transmission. Upon arriving at their destination, strings are parsed into (x, y) pairs or location data depending on if the information is coming to or from the frontend/backend of the system. https://github.com/BrockU-4P02-Logistics-System/RabbitMq/blob/main/src/main/java/ca/brocku/logistics/LogisticsApplication.java

5 Frontend

The frontend has three major sets of pages, a landing page, a set of authorization pages, and the user dashboard. The landing page is simple, a single page with several buttons concerned with redirecting the user to information within the same landing page as well as offering the key entry-point buttons, such as get started and create account. Both buttons serve as gateways to the authorization schema. These pages can be found within the following link https://github.com/BrockU-4P02-Logistics-System/Frontend/tree/main/app/auth. Once a login request is verified, this triggers the dashboard page to render for users, at which point users are greeted with the primary interface.

5.1 User Dashboard and page.tsx

The *meat and potatoes* of the frontend is *app/dashboard/page.tsx* (https://github.com/BrockU-4P02-Logistics-System/Frontend/blob/main/app/dashboard/page.tsx). This file contains all of the html for the buttons, text and visual components of the primary user interface. It is within this page that the user will enter stops, configure the number of drivers, change routing options, make routing requests, view the results as a geoline and see step by

step directions. Further, saving and reloading routes is facilitated from this page. The focus of the dashboard is an interactive Google Maps component. This component, took directly from the React Google maps library, is where we present location and geolines to the user once information is ready to display, residing in https://github.com/BrockU-4P02-Logistics-System/Frontend/tree/main/components/map.

Google maps also provides an auto complete component, which is utilized for users to add locations into the system efficiently. Once users have indicated an address to add via Google Maps auto complete, they add that stop to the route, and that location is displayed on the dashboard. The location is saved within the markers array, as a marker location object. Marker is a custom object that stores all necessary information regarding a location, namely;

- Address.
- Latitude and Longitude.
- Arrival Time.
- Departure Time.
- driverID.

Upon creation, arrival time and driverID are null, to be computed by the backend later.

In addition to passing the markers, route configuration must be passed too. Using the frontend interface, users will specify these parameters to pass to the backend, stored in the *routeConfiguration* object (return home, ferries...).

5.2 Optimize Route

The most critical function of the frontend, *calculateRoute*, is called when a user clicks the optimize route button. This triggers a chain of functions, sending the saved route configuration and nodes to the backend, handles the response, and then displays this information to the user. The initial steps of calculate route include credit handling, and a post call that sends the route data to the queue discussed in section 4. It then checks that the response is valid upon reception of said response. A valid response is then displayed along with its information to the user.

5.2.1 Process Driver Routes

The response that the frontend receives from the middleware is an array of routes, each route corresponding to one of the specified drivers. Each route is built of originally sent nodes, with the updated driverID field, previously null. A second array of markers is constructed from the routes data response. Once all of the markers have been updated, the next function is called, *processDriverRoutes*, which is passed the updated set of markers, *uniqueMarkers*. Process driver routes takes the updated markers and executes a for-each loop, in which driver

routes are processed to determine the route path and directions. This also includes error handling, such as unreachable and duplicate locations.

5.2.2 Route Path

A *routePath* is essential for drawing geolines (routePath is a data structure, an array of (x, y) pairs). This path is generated by the *getRoutePathFromDirections* function, responsible for taking the markers from an individual drivers route and establishing a connection with the Google Maps directions API. This API is passed the markers and route configuration settings, and returns an array of (x, y) coordinate pairs. Each of these pairs represents a vertex on the geoline to be displayed later. When the geoline is displayed, it simply renders a line between consecutive coordinate pairs in the response from the mentioned API.

5.2.3 Getting Directions

After the *routePath* has been completed, the directions are to be completed. The function *getDetailedDirections* works similarly to *getRoutePathDirections*, by sending marker locations and route configuration to a Google Maps API call, with the result being a list of step by step route instructions. Route distance and duration are within this function too.

The completion of all *routePaths* and *routeDirections* triggers page rendering. The Google Maps component now displays the geolines and directions in their appropriate areas of the dashboard. This component is interactive, and re-renders upon the altering of routes by users (adding/deleting nodes, configuration settings, etc). Since the user has the option to select a different drivers route, the *google.tsx* file automatically re-renders a driver directions and highlights the corresponding geoline.

5.3 Saving Loading Exporting

To service saving, loading and exporting routes, there are three main functions.

- *handleSavedRoute*
- *loadRoute*
- *handleExportDriver*

The function *handleSavedRoute* first checks for required credits, then ensures that the route has been calculated, stringifies that route's information, and sends to the database. The information being saved is an updated array of markers (driverID of route they belong to and order), configurations settings, and number of drivers. The function *loadRoute* is concerned with the opposite functionality. Route data is fetched from the database and then restores the markers array. The restored array is passed to *processDriverRoutes* exhibiting the same functionality as optimize route.

Finally, exporting routes is serviced by *handleExportDriver* that generates a

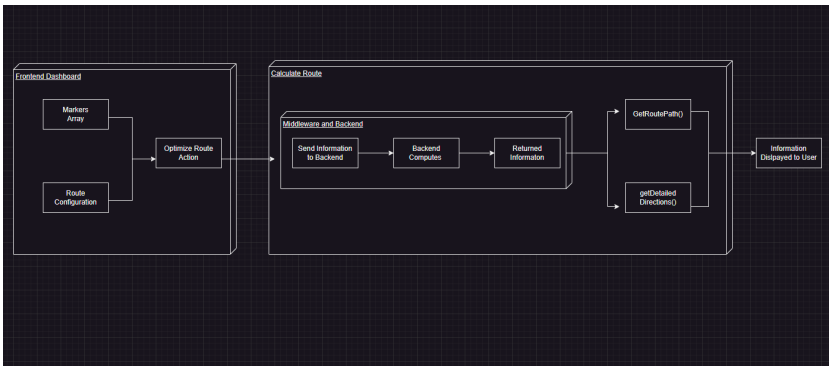

Figure 3: Flow of Front end Computation

Google Maps url with the *setMapurls* function. The url contains the marker data and configuration settings based on a driverID. This information is within that drivers corresponding *driverRoute* data structure.

The complete functionality of computing routes can be viewed in figure 3.

5.4 APIs

The frontend utilizes a collection of APIs. The endpoint between the frontend and the middleware is facilitated by the *route.tsx* file found by following https://github.com/BrockU-4P02-Logistics-System/Frontend/tree/main/app/api/process. The second API is for Stripe billing. To support the map, direction, and geolines components, we have the following Google API calls;

- **Map component:** Dynamically updates tiles to show geographical data and map.
- **Geocoding:** For the conversion of an address into coordinates for the backend.
- **Google Maps Directions Services Object:** Directions service object is used to get the directions for a list of ordered markers, and get (x, y) coordinates of those directions so that a geoline can be drawn by connecting them. It should be noted that we use Google Maps API objects to enhance the map, but these live on the client side, and do not require an API call. For example, getting detailed directions, an API call is made, but to display these directions, we use a client side object the *directionsRenderer*.

5.5 Database

For the code on session authorization and Mongoose connection, please follow https://github.com/BrockU-4P02-Logistics-System/Frontend/tree/main/lib. The database is concerned with storing information tied to users account as well as user authorization. The information that can be stored includes

routes, credits, and fleet details. In addition, the database communicates with the frontend on the web server for verification of log in and registration of new user accounts. The database communicates with the frontend when users want to save the information mentioned above to their accounts. The database has been created using MongoDB, as it can service the storage of larger files that can be tied to accounts such as GeoJson files for storing route information. MongoDB facilitates GeoJsons to be stored simply in BSON format. Finally, data sent to the database is schema validated to comply with the predefined formats.

For interaction with the fronend, the system makes use of Mongoose within its nextJS framework for facilitating the database connection with the webserver and sending objects. The database connection is opened via Mongoose and asynchronously verified. Mongoose is a Javascript library that uses predefined schema to allow object-oriented programming features to be used on database objects. Cyber security best principles are practiced by salting and hashing user passwords, which must abide by industry standard regular expressions, so that thry're never saved in plain text. Account admins have no way of viewing these passwords. The DB also has logic for preventing duplicate emails. The DB is hosted on the Oracle cloud server with the middleware and consumers.